WEST Search History

Hide Items | Restore | Clear | Cancel

DATE: Wednesday, March 01, 2006

Hide?	Set Name	Query	Hit Count
		DB=USPT; PLUR=NO; OP=OR	
□	L92	L91 and enumeration	0
□	L91	L90 and (stack near cache)	10
□	L90	L89 and (live near (object or objects))	110
□	L89	(l76 or l77 or l78 or l79 or l80 or L81) and (garbage adj1 collect$)	466
		DB=EPAB,JPAB,DWPI,TDBD; PLUR=NO; OP=OR	
□	L88	L86 and (stack near cache)	0
□	L87	L86 and enumeration	2
□	L86	L85 and (live near (object or objects))	17
□	L85	(garbage adj1 collect$)	1723
		DB=USPT; PLUR=NO; OP=OR	
□	L84	(l76 or l77 or l78 or l79 or l80 or L81) and (enumeration with cache)	5
□	L83	L82 and (enumeration with cache)	0
□	L82	(l76 or l77 or l78 or l79 or l80 or L81) and (stack with cache)	102
□	L81	711/118.ccls.	1316
□	L80	711/113.ccls.	759
□	L79	711/6.ccls.	141
□	L78	707/103r-103z.ccls.	1177
□	L77	707/206.ccls.	470
□	L76	707/200-201.ccls.	2170
□	L75	L64 and stack	0
□	L74	L64 and (enumeration near stack)	0
□	L73	L64 and enumeration	1
□	L72	L64 and (root near enumeration)	0
□	L71	L64 and (root near enumeration near stack)	0
□	L70	L64 and (root with enumeration with stack)	0
□	L69	L64 and (root with enumeration with stack with cache)	0
□	L68	L64 and (root near enumeration near stack near cache)	0
□	L67	L64 and ((root adj1 set) near enumeration near stack near cache)	0
□	L66	L64 and ((root adj1 set) with enumeration with stack with cache)	0
□	L65	L64 and ((root adj1 set) with enumeration with stack with trace)	0

10/632,474

□	L64	6978285.pn.	1
		DB=PGPB,USPT; PLUR=NO; OP=OR	
□	L63	L62 and (garbage adj1 collect$)	4
□	L62	(enumeration near list)	110
□	L61	(enumeration near stack)	2
□	L60	(enumeration with stack)	44
		DB=PGPB; PLUR=NO; OP=OR	
□	L59	L58 and compiler	45
□	L58	L55 and virtual	49
□	L57	L56 and enumeration	3
□	L56	L55 not intel	35
□	L55	(live with (object or objects) with (root adj1 (set or sets)))	50
		DB=USPT; PLUR=NO; OP=OR	
□	L54	L40 not intel	24
□	L53	L52 and compil$	21
□	L52	L50 and virtual	26
□	L51	L50 and enumeration	0
□	L50	L49 not intel	33
□	L49	L48 and (garbage adj1 collect$)	39
□	L48	((live near (object or objects)) with root)	40
□	L47	((love near (object or objects)) with root)	0
□	L46	L29 and thread	6
□	L45	L37 and L41	3
□	L44	L36 and L41	0
□	L43	L37 and L40	27
□	L42	L36 and L40	17
□	L41	(live near (object or objects) near (root adj1 (set or sets)))	3
□	L40	(live with (object or objects) with (root adj1 (set or sets)))	30
□	L39	L37 and enumeration	6
□	L38	L36 and enumeration	3
□	L37	(garbage adj1 collect$).ab.	287
□	L36	(garbage adj1 collect$).ti.	179
□	L35	L33 and (garbage near collect$)	0
□	L34	(stack near cach$ near thread)	0
□	L33	(stack with cach$ with thread)	9
□	L32	L31 and (garbage near collect$)	2
□	L31	(enumeration with stack)	23
□	L30	(enumeration near stack)	1

□	L29	L27 and (stack with thread)	6
□	L28	L27 and (stack near thread)	0
□	L27	L26 and (stack near cach$)	16
□	L26	(6415302 6424977 6434576 6434577 6449626 6381738 6748503 5241673 5485613 4907151 5848423 5893121 5930807 6199075 6253215 6279012 6421689 6842853 6904589 6907437 5369732 6065020 6978285 6951018 5392432 5900001 5903900 5911144 5915255 5920876 6038572 6049810 6094664 6115782 6308319 6594820 6598141 6618738 6662274 6898611 6912553 5218698 6081665 4887235 4922414 4951194 6327701 6829686 6925637 5652883).pn. (5317764 5664060 6070173 6799253 6282702 6429302 4816711 6215907 6215907 5680509 6185581 5953736 4797810 6098089 5321834 5577246 6093216 6148310 6173294 6286016 6308185 6317869 6453403 6470361 6529919 6766336 6868488 6965905 6047295 6098080 5560003 5692185 5819304 5590332 6047125 5903899 5970781 4912629 5432908 5566321 5605231 5613345 5799324 5909579 5918235 6055612 6101580 6125434 6192517 6237009).pn. (6247026 6275985 6289360 6304949 6308315 6381735 6442663 6442751 6449625 6457023 6557091 6625808 6675379 6681385 6691306 6718539 6735680 6735761 6738846 6769004 6792601 6799191 6804681 6836782 6845437 6862674 6934726 6957422 6990567 4161252 5305587 5816771 5819255 5925123 6205441 6330659 6438741 4568163 4600680 4909779 5322180 5983259 4290530 4417801 5285249 5521777 6155398 6167766 5003470 5664086).pn. (5949435 6341284 6493663 5220665 5649027 5752027 5913206 5913207 5960423 6012103 6249825 6278997 6262984 4482297 4482296 4848976 4907180 5284406 5426890 5528508 5625812 5765014 5953527 5701470 5761670 6016508 4755952 4811253 5007135 5201512 5675755 5729901 6000475 6010304 6031530 6086643 6379107 6405305 6409449 6466984 4601905 5933611 6081838 5845298 6144965 6249793 4961137 5355483 5819299 5860135).pn. (5960087 5991779 6272504 6314436 6324631 6338073 6343296 6349314 6349334 6393439 6421660 6427154 6430580 6502110 6502111 6526422 6560619 6594749 6671707 6701520 6795836 6823351 6826583 6839726 6879991 6889303 6931423 6950838 5607496 6038643 3554143 3651771 3881431 3863577 3815523 4064318 4144824 4245790 4246850 4285719 4347632 4474108 4624095 4793270 4817539 4829911 4924762 5048764 5183157 5191846).pn. (5199362 5255967 5289786 5328088 5366169 5474229 5535576 5542306 5551565 5699745 5753012 5787431 5799597 5803299 5921323 5926933 5940621 RE36553 6026237 6105859 6112823 6151703 6180396 6185862 6189460 6193503 6237043 6237060 6237135 6247020 RE37350 6480507 6505344 6658653 6675378 6839725 6980997 6983357 6373485 6468523 4626314 5961454 3908969 4248209 4293214 4353542 4358197 4370710 4411515 4558942).pn.	299
□	L25	((stack near cach$) near (stack near thread))	0
□	L24	((stack near cach$) with (stack near thread))	0
□	L23	L22 and (garbage near collect$)	0
□	L22	((stack with cach$) with (stack with thread))	9
□	L21	L20 and (live near (object or objects))	2
□	L20	L19 and (garbage near collect$)	27

□	L19	(L17 and L18)	55
□	L18	(stack with cach$)	1033
□	L17	(stack with thread)	962
□	L16	L9 and (live near (object or objects))	0
□	L15	L11 and (live near (object or objects))	17
□	L14	L13 and (live near (object or objects))	3
□	L13	L12 and (garbage near collect$)	51
□	L12	(stack near trac$)	539
□	L11	L10 and (garbage near collect$)	100
□	L10	(stack near thread)	249
□	L9	L8 and (garbage near collect$)	21
□	L8	(trac$ near cach$)	750
□	L7	L6 and (garbage near collect$)	30
□	L6	L4 and thread$	40
□	L5	(L2 and L4)	0
□	L4	(trac$ with live with object$)	97
□	L3	(trac$ with object$)	55746
□	L2	(stack with trace with cach$)	5
□	L1	(stack near trace near cach$)	0

END OF SEARCH HISTORY

Dialog DataStar

options | logoff | feedback | help

databases | easy search

Advanced Search:

Inspec - 1969 to date (INZZ)

limit

Search history:

No.	Database	Search term	Info added since	Results	
1	INZZ	garbage ADJ collect$	unrestricted	88	show titles
2	INZZ	1 AND enumerat$	unrestricted	0	-
3	INZZ	1 AND stack WITH cache	unrestricted	0	-
4	INZZ	1 AND live NEAR (object OR objects)	unrestricted	2	show titles
5	INZZ	1 AND stack	unrestricted	9	show titles
6	INZZ	5 AND cach$	unrestricted	0	-

hide | delete all search steps... | delete individual search steps...

Enter your search term(s): Search tips □ Thesaurus mapping

whole document

Information added since: or: none
(YYYYMMDD)

search

Select special search terms from the following list(s):

- Publication year
- Inspec thesaurus - browse headings A-G
- Inspec thesaurus - browse headings H-Q
- Inspec thesaurus - browse headings R-Z
- Inspec thesaurus - enter a term
- Classification codes A: Physics, 0-1
- Classification codes A: Physics, 2-3
- Classification codes A: Physics, 4-5
- Classification codes A: Physics, 6
- Classification codes A: Physics, 7
- Classification codes A: Physics, 8

10/632,474

Dialog DataStar

options logoff feedback help

database search page titles

Document

Select the documents you wish to save or order by clicking the box next to the document, or click the link above the document to order directly.

save locally as: PDF document search strategy: do not include the search strategy

order

☑ Select All

1 Estimating the impact of heap liveness information on space consumpti
2 Contaminated garbage collection.
3 Reducing garbage in Java.
4 New computation model, queue machine, and its application to parallel functiona
5 CONS should not CONS its arguments. II. Cheney on the M.T.A.
6 Lambda-calculus schemata.
7 An abstract machine design for lexically scoped parallel Lisp with speculati
8 Memory allocation and higher-order functions.
9 Lambda calculus schemata.

☑ **document 1 of 9** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update
0007945373 20051201.

Title
Estimating the impact of heap liveness information on space consumption in Java.

Conference information
ISMM'02: International Symposium on Memory Management, Berlin, Germany, 20-21 June 2002.

Source
SIGPLAN Notices, {SIGPLAN-Not-USA}, Feb. 2003, p. 171-82, 12 refs, CODEN: SINODQ, ISSN: 0362-1340.
Publisher: ACM, USA.

Author(s)
Shaham-R, Kolodner-E-K, Sagiv-M.

Author affiliation
Shaham, R., Tel Aviv Univ., Israel.

Abstract
We study the potential impact of different kinds of liveness information on the space consumption of a program in a garbage collected environment, specifically for Java. The idea is to measure the time difference between the actual time an object is collected by the garbage collector (GC) and the potential earliest time an object could be collected assuming liveness information were available. We focus on the following kinds of liveness information: (i) stark reference liveness (local reference variable liveness in Java), (ii) global reference liveness (static reference variable liveness in Java) , (iii) heap reference liveness (instance reference variable liveness or array reference liveness in Java), and (vi) any combination of (i)-(iii). We also provide some insights on the kind of interface between a compiler and GC that could achieve these potential savings. The Java Virtual Machine (JVM) was instrumented to measure (dynamic) liveness information. Experimental results are given for 10 benchmarks, including 5 of the SPEC-jvm98 benchmark suite. We show that in general stack reference

liveness may yield small benefits, global reference liveness combined with stack reference liveness may yield medium benefits, and heap reference liveness yields the largest potential benefit. Specifically, for heap reference liveness we measure an average potential savings of 39% using an interface with complete liveness information, and an average savings of 15% using a more restricted interface.

Descriptors

DATA-STRUCTURES; JAVA; PROGRAM-COMPILERS; PROGRAM-DIAGNOSTICS; SOFTWARE-PERFORMANCE-EVALUATION; STORAGE-MANAGEMENT.

Classification codes

C6120 File-organisation*;
C6150G Diagnostic-testing-debugging-and-evaluating-systems;
C6140D High-level-languages;
C6150C Compilers-interpreters-and-other-processors.

Keywords

heap-liveness-information; space-consumption; garbage-collection; stark-reference-liveness; local-reference-variable-liveness; global-reference-liveness; static-reference-variable-liveness; heap-reference-liveness; instance-reference-variable-liveness; array-reference-liveness; compiler-interface; memory-management; program-analysis; Java-Virtual-Machine; JVM; dynamic-liveness; SPEC-jvm98-benchmark-suite.

Treatment codes

P Practical.

Language

English.

Publication type

Conference-proceedings; Journal-paper.

Availability

SICI: 0362-1340(200302)+L.171:EIHL; 1-I.

Publication year

2003.

Publication date

20030200.

Edition

2004017.

Copyright statement

Copyright 2004 IEE.

USPTO Full Text Retrieval Options

☑ **document 2 of 9** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update

0006656143 20051201.

Title

Contaminated garbage collection.

Conference information

ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PDLI), Vancouver, BC, Canada, 18-21 June 2000.
Sponsor(s): ACM.

Source

SIGPLAN Notices, {SIGPLAN-Not-USA}, May 2000, vol. 35, no. 5, p. 264-73, 20 refs, CODEN: SINODQ, ISSN: 0362-1340.
Publisher: ACM, USA.

Author(s)

Cannarozzi-D-J, Plezbert-M-P, Cytron-R-K.

Author affiliation

Cannarozzi, D.J., Plezbert, M.P., Cytron, R.K., Dept. of Comput. Sci., Washington Univ., St. Louis, MO, USA.

Abstract

We describe a new method for determining when an object can be garbage collected. The method does not require marking live objects. Instead, each object X is dynamically associated with a stack frame M, such that X is collectable when M pops. Because X could have been dead earlier, our method is conservative. Our results demonstrate that the method nonetheless identifies a large percentage of collectable objects. The method has been implemented in Sun's Java/sup TM/ Virtual Machine interpreter, and results are presented based an this implementation.

Descriptors

JAVA; PROGRAM-INTERPRETERS; STORAGE-MANAGEMENT.

Classification codes

C6120 File-organisation*;
C6150C Compilers-interpreters-and-other-processors;
C6110J Object-oriented-programming;
C6140D High-level-languages.

Keywords

contaminated-garbage-collection; stack-frame; collectable-objects; Sun-Java-Virtual-Machine-interpreter.

Treatment codes

P Practical.

Language

English.

Publication type

Conference-proceedings; Journal-paper.

Availability

SICI: 0362-1340(200005)35:5L.264:CGC; 1-6.

Publication year

2000.

Publication date

20000500.

Edition

2000029.

Copyright statement

USPTO Full Text Retrieval Options

☑ **document 3 of 9** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update

0006060712 20051201.

Title

Reducing garbage in Java.

Source

SIGPLAN Notices, {SIGPLAN-Not-USA}, Sept. 1998, vol. 33, no. 9, p. 84-6, 2 refs, CODEN: SINODQ, ISSN: 0362-1340.
Publisher: ACM, USA.

Author(s)

McDowell-C-E.

Author affiliation

McDowell, C.E., Dept. of Comput. & Inf. Sci., California Univ., Santa Cruz, CA, USA.

Abstract

One of the important advantages of Java, from a programmers prospective, is the use of garbage collection. One aspect of memory management in Java is that all objects are created on a garbage collected heap. Only primitive types, mostly numeric types and references to objects, are allocated on the runtime stack. The author speculated that a significant number of objects behaved like traditional automatic variables, that are normally allocated on the runtime stack. The author instrumented a Java virtual machine to test this hypothesis. The percentage of objects that could have been allocated on a stack instead of on the heap ranged from zero to possibly as high as 56%, but were generally in the 5-15% range.

Descriptors

ABSTRACT-DATA-TYPES; OBJECT-ORIENTED-PROGRAMMING; STORAGE-MANAGEMENT.

Classification codes

C6120 File-organisation*;
C6110J Object-oriented-programming.

Keywords

garbage-reduction; Java; garbage-collection; memory-management; garbage-collected-heap; primitive-type-allocation; numeric-types; object-references; runtime-stack; automatic-variables; Java-virtual-machine.

Treatment codes

P Practical.

Language

English.

Publication type

Journal-paper.

Availability

SICI: 0362-1340(199809)33:9L.84:RGJ; 1-H.

Publication year

1998.

Publication date

19980900.

Edition

1998042.

Copyright statement

Copyright 1998 IEE.

USPTO Full Text Retrieval Options

☑ **document 4 of 9** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update

0005601243 20051201.

Title

New computation model, queue machine, and its application to parallel functional programming languages.

Source

Transactions of the Information Processing Society of Japan, {Trans-Inf-Process-Soc-Jpn-Japan}, March 1997, vol. 38, no. 3, p. 574-83, 20 refs, CODEN: JSGRD5, ISSN: 0387-5806.
Publisher: Inf. Process. Soc. Japan, Japan.

Author(s)

Maeda-A, Nakanishi-M.

Abstract

The authors present a new evaluation scheme for expressions called queue machine model of execution which enables automatic (implicit) parallel execution of functional programming languages with very small synchronization overhead without special hardware support. In purely functional languages, multiple function call can be evaluated parallelly without changing the semantics of the program. But when implemented naively, synchronization overhead to wait for termination of all subcomputations becomes prohibitive. Moreover, local context information usually stored in a stack must be maintained in a garbage-collected heap. So overhead of memory management also increases when compared to sequential implementations. They show that by emulating execution model of queue machines and by replacing stacks with queues, the overhead can be drastically reduced and parallel function invocation can be implemented efficiently on stock hardware. Preliminary measurement of prototype implementation based on this technique is presented. The measurement shows that, although programs compiled with their prototype compiler run slower than other implementations on sequential machines, they show good scalability and run faster than sequential implementations when executed with two or more processors.

Descriptors

FUNCTIONAL-LANGUAGES; PARALLEL-LANGUAGES; PROGRAM-COMPILERS; STORAGE-MANAGEMENT; SYNCHRONISATION.

Classification codes

C6140D High-level-languages*;
C6120 File-organisation;
C6150C Compilers-interpreters-and-other-processors.

Keywords

queue-machine-model; parallel-functional-programming-languages; computation-model; automatic-parallel-execution; synchronization-overhead; parallel-multiple-function-call-evaluation; purely-functional-languages; subcomputation-termination; local-context-information; garbage-collected-heap; memory-management-overhead; execution-model-emulation; parallel-function-invocation; stock-hardware; scalability; compiler.

Treatment codes

T Theoretical-or-mathematical.

Language

Japanese.

Publication type

Journal-paper.

Availability

SICI: 0387-5806(199703)38:3L.574:CMQM; 1-D.

Publication year

1997.

Publication date

19970300.

Edition

1997023.

Copyright statement

Copyright 1997 IEE.

USPTO Full Text Retrieval Options

☑ **document 5 of 9** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update

0005085135 20051201.

Title

CONS should not CONS its arguments. II. Cheney on the M.T.A.

Source

SIGPLAN Notices, {SIGPLAN-Not-USA}, Sept. 1995, vol. 30, no. 9, p. 17-20, 21 refs, CODEN: SINODQ, ISSN: 0362-1340, USA.

Author(s)

Baker-H-G.

Abstract

For pt.I, see ibid., vol.27, no.3, p.24-34, 1992. Previous Schemes for implementing full tail recursion when compiling into C have required some form of "trampoline' to pop the stack. We propose solving the tail recursion problem in the same manner as Standard ML of New Jersey, by allocating all frames in the (garbage collected) heap. The Scheme program is translated into continuation passing style, so the target C functions never return. The C stack pointer then becomes the allocation pointer for a Cheney style copying garbage collection scheme. Our Scheme can use C function calls, C arguments, C variable arity functions, and separate compilation without requiring complex block compilation of entire programs.

Descriptors

C-LANGUAGE; DATA-STRUCTURES; PROGRAM-COMPILERS.

Classification codes

C6140D High-level-languages*;
C6120 File-organisation;
C6150C Compilers-interpreters-and-other-processors.

Keywords

CONS; Cheney; Scheme; full-tail-recursion; tail-recursion-problem; Standard-ML; garbage-collected-heap; Scheme-program; continuation-passing-style; target-C-functions; C-stack-pointer; allocation-pointer; Cheney-style-copying-garbage-collection-scheme; C-function-calls; C-arguments; C-variable-arity-functions.

Treatment codes

P Practical.

Language

English.

Publication type

Journal-paper.

Publication year

1995.

Publication date

19950900.

Edition

1995041.

Copyright statement

Copyright 1995 IEE.

USPTO Full Text Retrieval Options

☑ **document 6 of 9** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update

0004582180 20051201.

Title

Lambda-calculus schemata.

Source

LISP and Symbolic Computation, {LISP-Symb-Comput-Netherlands}, Nov. 1993, vol. 6, no. 3-4, p. 259-88, 37 refs, CODEN: LSCOEX, ISSN: 0892-4635, Netherlands.

Author(s)

Fischer-M-J.

Author affiliation

Fischer, M.J., Dept. of Comput. Sci., Yale Univ., New Haven, CT, USA.

Abstract

A lambda-calculus schema is an expression of the lambda calculus augmented by uninterpreted constant and operator symbols. It is an abstraction of programming languages such as LISP which permit functions to be passed to and returned from other functions. When given an interpretation for

its constant and operator symbols, certain schemata, called lambda abstractions, naturally define partial functions over the domain of interpretation. Two implementation strategies are considered: the retention strategy in which all variable bindings are retained until no longer needed (implying the use of some sort of garbage-collected store) and the deletion strategy, modeled after the usual stack implementation of ALGOL 60, in which variable bindings are destroyed when control leaves the procedure (or block) in which they were created. Not all lambda abstractions evaluate correctly under the deletion strategy. Nevertheless, both strategies are equally powerful in the sense that any lambda abstraction can be mechanically translated into another that evaluates correctly under the deletion strategy and defines the same partial function over the domain of interpretation as the original. Proof is by translation into continuation-passing style.

Descriptors
FORMAL-LANGUAGES; LAMBDA-CALCULUS; LISP.

Classification codes
C4210 Formal-logic*;
C6140D High-level-languages.

Keywords
lambda-calculus-schema; uninterpreted-constant-symbols; uninterpreted-operator-symbols; programming-languages; LISP; lambda-abstractions; partial-functions; implementation-strategies; retention-strategy; variable-bindings; garbage-collected-store; deletion-strategy; stack-implementation; ALGOL-60; continuation-passing-style.

Treatment codes
T Theoretical-or-mathematical.

Language
English.

Publication type
Journal-paper.

Availability
CCCC: 0892-4635/93/$5.00.

Publication year
1993.

Publication date
19931100.

Edition
1994002.

Copyright statement
Copyright 1994 IEE.

USPTO Full Text Retrieval Options

☑ **document 7 of 9** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update
0004314574 20051201.

Title
An abstract machine design for lexically scoped parallel Lisp with speculative processing.

Source
SIGPLAN Notices, {SIGPLAN-Not-USA}, Nov. 1992, vol. 27, no. 11, p. 77-84, 14 refs, CODEN: SINODQ, ISSN: 0362-1340, USA.

Author(s)
Yuen-C-K.

Author affiliation
Yuen, C.K., DISCS, Nat. Univ. of Singapore, Singapore.

Abstract

An abstract machine is designed to support the data environment requirements of Balinda Lisp, a

parallel Lisp dialect which permits speculative processing of conditional modules. Logically, the machine provides multiple stacks connected into an environment tree, with lexically visible sections pointed to by display registers. Physically, stack sections are stored as separate objects and access is established by the use of a dynamic stack recording the chain of function calls, and a set of lexical display registers pointing at visible objects. This arrangement allows parts of the environment of a function to be retained or garbage-collected as appropriate after exit. By making copies of visible ancestral stack sections, side effects of speculative parallel tasks are handled in accordance with language semantics. The architecture is generic and may be realized in a variety of forms, depending on whether BaLinda Lisp is implemented on a conventional machine, stack machine, or dataflow machine.

Descriptors

DATA-STRUCTURES; LISP; PARALLEL-LANGUAGES; PARALLEL-MACHINES.

Classification codes

C6140D High-level-languages*;
C6110P Parallel-programming;
C6120 File-organisation;
C5220P Parallel-architecture.

Keywords

abstract-machine-design; lexically-scoped-parallel-Lisp; data-environment-requirements; Balinda-Lisp; parallel-Lisp-dialect; speculative-processing; conditional-modules; multiple-stacks; environment-tree; lexically-visible-sections; display-registers; stack-sections; dynamic-stack; function-calls; lexical-display-registers; visible-objects; garbage-collected; visible-ancestral-stack-sections; speculative-parallel-tasks; language-semantics.

Treatment codes

P Practical.

Language

English.

Publication type

Journal-paper.

Publication year

1992.

Publication date

19921100.

Edition

1992056.

Copyright statement

USPTO Full Text Retrieval Options

☑ **document 8 of 9** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update

0003046622 20051201.

Title

Memory allocation and higher-order functions.

Conference information

SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, St. Paul, MN, USA, 24-26 June 1987.
Sponsor(s): ACM; IEEE.

Source

SIGPLAN Notices, {SIGPLAN-Not-USA}, July 1987, vol. 22, no. 7, p. 241-52, 38 refs, CODEN: SINODQ, ISSN: 0362-1340, USA.

Author(s)

Danvy-O.

Author affiliation

Danvy, O., Inst. of Datalogy, Copenhagen Univ., Denmark.

Abstract

Presents a constant-time marking-collecting algorithm to efficiently implement recursion with a general heap memory rather than with a vectorial stack, in a context of frequent captures of continuations. It has been seen to reduce the 80% garbage collection overhead to less than 5% on average. The algorithm has been built into a virtual machine to efficiently implement at the assembly level the actor language PLASMA, an actor-oriented version of PROLOG and variant of Scheme, currently in use on 8086, 68000 and VAX. The rationale to use the heap memory is that continuations are available via a single pointer in a unified memory and can be shared optimally when recurrently captured, which is simply impossible using a strategy based on stack recopy. Further, non-captured continuations can be incrementally garbage collected on the fly. The author describes the elementary recursive instructions of the virtual machine, presents and proves the marking-collecting strategy, and safely generalizes the transformation `call + return = branch' in a way compatible with the possible capture of the current continuation. An appendix relates its integration in the `Virtual Scheme Machine' supporting Scheme 84.

Descriptors

STORAGE-MANAGEMENT; VIRTUAL-MACHINES.

Classification codes

C6120 File-organisation*;
C7430 Computer-engineering.

Keywords

memory-allocation; frequent-continuation-captures; noncaptured-recursive-contexts; functional-languages; optimal-sharing; incremental-collection; higher-order-functions; constant-time-marking-collecting-algorithm; recursion; general-heap-memory; garbage-collection-overhead; virtual-machine; assembly-level; actor-language; PLASMA; pointer.

Treatment codes

P Practical.

Language

English.

Publication type

Conference-proceedings; Journal-paper.

Availability

CCCC: 0362-1340/87/0006/0241$00.75.

Publication year

1987.

Publication date

19870700.

Edition

1988003.

Copyright statement

☑ **document 9 of 9** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update

0000424715 20051201.

Title

Lambda calculus schemata.

Conference information

Proceedings of an ACM Conference on Proving Assertations about Programs, Las Cruces, NM, USA, 6-7 Jan. 1972.
Sponsor(s): ACM.

Source

Proceedings of an ACM Conference on Proving Assertations about Programs, 1972, p. 104-9, 14 refs, pp. iv+211.
Publisher: ACM, New York, NY, USA.

Author(s)

Fischer-M-J.

Author affiliation

Fischer, M.J., MIT, Cambridge, MA, USA.

Abstract

Considers two natural implementation strategies: the retention strategy in which all variable bindings are retained until no longer needed (implying the use of some sort of garbage collected store) and the deletion strategy, modelled after the usual stack implementation of ALGOL-60, in which variable bindings are destroyed when control leaves the procedure (or block) in which they were created.

Descriptors

COMPUTATION-THEORY.

Classification codes

C4290 Other-computer-theory*.

Keywords

lambda-calculus-schemata; retention-strategy; implementation-strategies; deletion-strategy.

Treatment codes

T Theoretical-or-mathematical.

Language

English.

Publication type

Conference-proceedings.

Publication year

1972.

Publication date

19720000.

Edition

1972008.

Copyright statement

locally as: PDF document search strategy: do not include the search strategy

Top - News & FAQS - Dialog

Dialog DataStar

options logoff feedback help

databases search page titles

Document

Select the documents you wish to save or order by clicking the box next to the document, or click the link above the document to order directly.

save locally as: PDF document search strategy: do not include the search strategy

order

☑ Select All

1 A performance analysis of the active memory system.

2 Contaminated garbage collection.

☑ **document 1 of 2** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update

0007093373 20051201.

Title

A performance analysis of the active memory system.

Conference information

Proceedings 2001 International Conference on Computer Design. ICCD 2001, Austin, TX, USA, 23-26 Sept. 2001.

Sponsor(s): IEEE Comput. Soc; IEEE Circuits & Syst. Soc; IEEE Electron Devices Soc.

Source

Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001, 2001, p. 493-6, 10 refs, pp. xxii+559, ISBN: 0-7695-1200-3.

Publisher: IEEE Comput. Soc, Los Alamitos, CA, USA.

Author(s)

Witawas-Srisa-An, Srisa-an, Chia-Tien-Dan-Lo, J-Morris-Chang.

Author affiliation

Witawas Srisa-An, Srisa-an, Chia-Tien Dan Lo, J Morris Chang, Dept. of Comput. Sci., Illinois Inst. of Technol., Chicago, IL, USA.

Abstract

One major problem of using Java in real-time and embedded devices is the non-deterministic turnaround time of dynamic memory management systems (memory allocation and garbage collection). For the allocation, the nondeterminism is often contributed by the time to perform searching, splitting, and coalescing. For the garbage collection, the turnaround time is usually determined by the size of the heap, the number of live objects, the number of object collected, and the amount of garbage collected Even with the current state-of-the-art garbage collectors (generational and incremental schemes), they may or may not guarantee the worst case latency. Moreover such schemes often prolong overall garbage collection time. In this paper, the performance analysis of the proposed Active Memory Module (AMM) for embedded systems is presented Unlike the software counterparts, the AMM can perform a memory allocation in a predictable and hounded fashion (14 cycles). Moreover it can also yield a bounded sweeping time regardless of the number of live objects or heap size. By utilizing the proposed system, the overall speed-up can be as high as 23% over the JDK 1.2.2 running in classic mode.

Descriptors

JAVA; REAL-TIME-SYSTEMS; STORAGE-ALLOCATION; STORAGE-MANAGEMENT.

Classification codes

C6120 File-organisation*;
C6110J Object-oriented-programming;
C6140D High-level-languages;
C6150N Distributed-systems-software.

Keywords

Performance-Analysis; Active-Memory-System; Java; embedded-devices; real-time-devices; dynamic-memory-management-systems; worst-case-latency; memory-allocation; garbage-collection; nondeterminism.

Treatment codes

A Application;
P Practical.

Language

English.

Publication type

Conference-proceedings.

Availability

CCCC: 0-7695-1200-3/01/$10.00.

Digital object identifier

10.1109/ICCD.2001.955073.

Publication year

2001.

Publication date

20010000.

Edition

2001045.

Copyright statement

Copyright 2001 IEE.

USPTO Full Text Retrieval Options

☑ **document 2 of 2** Order Document

Inspec - 1969 to date (INZZ)

Accession number & update

0006656143 20051201.

Title

Contaminated garbage collection.

Conference information

ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PDLI), Vancouver, BC, Canada, 18-21 June 2000.
Sponsor(s): ACM.

Source

SIGPLAN Notices, {SIGPLAN-Not-USA}, May 2000, vol. 35, no. 5, p. 264-73, 20 refs, CODEN: SINODQ, ISSN: 0362-1340.
Publisher: ACM, USA.

Author(s)

Cannarozzi-D-J, Plezbert-M-P, Cytron-R-K.

Author affiliation

Cannarozzi, D.J., Plezbert, M.P., Cytron, R.K., Dept. of Comput. Sci., Washington Univ., St. Louis, MO, USA.

Abstract

We describe a new method for determining when an object can be garbage collected. The method does not require marking live objects. Instead, each object X is dynamically associated with a stack frame

M, such that X is collectable when M pops. Because X could have been dead earlier, our method is conservative. Our results demonstrate that the method nonetheless identifies a large percentage of collectable objects. The method has been implemented in Sun's Java/sup TM/ Virtual Machine interpreter, and results are presented based an this implementation.

Descriptors

JAVA; PROGRAM-INTERPRETERS; STORAGE-MANAGEMENT.

Classification codes

C6120 File-organisation*;
C6150C Compilers-interpreters-and-other-processors;
C6110J Object-oriented-programming;
C6140D High-level-languages.

Keywords

contaminated-garbage-collection; stack-frame; collectable-objects; Sun-Java-Virtual-Machine-interpreter.

Treatment codes

P Practical.

Language

English.

Publication type

Conference-proceedings; Journal-paper.

Availability

SICI: 0362-1340(200005)35:5L.264:CGC; 1-6.

Publication year

2000.

Publication date

20000500.

Edition

2000029.

Copyright statement

Copyright 2000 IEE.

save locally as: PDF document search strategy: do not include the search strategy

order

Top - News & FAQS - Dialog

PORTAL
USPTO

Subscribe (Full Service) Register (Limited Service, Free) Login

Search: ● The ACM Digital Library ○ The Guide

live objects and garbage collection and stack and cache and ro

THE ACM DIGITAL LIBRARY

Feedback Report a problem Satisfaction survey

Terms used
live objects and **garbage collection** and **stack** and **cache** and **root**

Found **28,219** of **171,143**

Sort results by: relevance
Display results: expanded form

Save results to a Binder
Search Tips
□ Open results in a new window

Try an Advanced Search
Try this search in The ACM Guide

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next

Best 200 shown

Relevance scale □ □ □ □ □

1 Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign
Chen-Yong Cher, Antony L. Hosking, T. N. Vijaykumar
October 2004 **ACM SIGOPS Operating Systems Review , ACM SIGPLAN Notices , ACM SIGARCH Computer Architecture News , Proceedings of the 11th international conference on Architectural support for programming languages and operating systems ASPLOS-XI**, Volume 38 , 39 , 32 Issue 5 , 11 , 5
Publisher: ACM Press
Full text available: pdf(165.32 KB) Additional Information: full citation, abstract, references, index terms

Tracing garbage collectors traverse references from live program variables, transitively tracing out the closure of live objects. Memory accesses incurred during tracing are essentially random: a given object may contain references to any other object. Since application heaps are typically much larger than hardware caches, tracing results in many cache misses. Technology trends will make cache misses more important, so tracing is a prime target for prefetching.Simulation of Java benchmarks runni ...

Keywords: breadth-first, buffered prefetch, cache architecture, depth-first, garbage collection, mark-sweep, prefetch-on-grey, prefetching

2 Garbage collection for a client-server persistent object store
Laurent Amsaleg, Michael J. Franklin, Olivier Gruber
August 1999 **ACM Transactions on Computer Systems (TOCS)**, Volume 17 Issue 3
Publisher: ACM Press
Full text available: pdf(267.18 KB) Additional Information: full citation, abstract, references, citings, index terms, review

We describe an efficient server-based algorithm for garbage collecting persistent object stores in a client-server environmnet. The algorithm is incremental and runs concurrently with client transactions. Unlike previous algorithms, it does not hold any transactional locks on data and does non require callbacks to clients. It is fault-tolerant, but performs very little logging. The algorithm has been designed to be integrated into existing systems, and therefore it works with standard i ...

Keywords: client-server system, logging, persistent object-store, recovery

10\632,474

3 A unified theory of garbage collection
David F. Bacon, Perry Cheng, V. T. Rajan
October 2004 **ACM SIGPLAN Notices , Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications OOPSLA '04**, Volume 39 Issue 10
Publisher: ACM Press
Full text available: pdf(223.52 KB) Additional Information: full citation, abstract, references, index terms

Tracing and reference counting are uniformly viewed as being fundamentally different approaches to garbage collection that possess very distinct performance properties. We have implemented high-performance collectors of both types, and in the process observed that the more we optimized them, the more similarly they behaved - that they seem to share some deep structure.

We present a formulation of the two algorithms that shows that they are in fact duals of each other. Intuitively, the ...

Keywords: graph algorithms, mark-and-sweep, reference counting, tracing

4 Generational stack collection and profile-driven pretenuring
Perry Cheng, Robert Harper, Peter Lee
May 1998 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation PLDI '98**, Volume 33 Issue 5
Publisher: ACM Press
Full text available: pdf(1.56 MB) Additional Information: full citation, abstract, references, citings, index terms

This paper presents two techniques for improving garbage collection performance: generational stack collection and profile-driven pretenuring. The first is applicable to stack-based implementations of functional languages while the second is useful for any generational collector. We have implemented both techniques in a generational collector used by the TIL compiler (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996), and have observed decreases in garbage collection times of as much as 70 ...

5 Efficient memory management in a merged heap/stack prolog machine
Xining Li
September 2000 **Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming**
Publisher: ACM Press
Full text available: pdf(553.36 KB) Additional Information: full citation, references, index terms

6 Comparing mostly-copying and mark-sweep conservative collection
Frederick Smith, Greg Morrisett
October 1998 **ACM SIGPLAN Notices , Proceedings of the 1st international symposium on Memory management ISMM '98**, Volume 34 Issue 3
Publisher: ACM Press
Full text available: pdf(1.52 MB) Additional Information: full citation, abstract, references, citings, index terms

Many high-level language compilers generate C code and then invoke a C compiler for code generation. To date, most, of these compilers link the resulting code against a conservative mark-sweep garbage collector in order to reclaim unused memory. We introduce a new collector, MCC, based on an extension of *mostly-copying collection*.We analyze the various design decisions made in MCC and provide a performance comparison to the most widely used conservative mark-sweep collector (the Boehm-Dem ...

7 A parallel, incremental, mostly concurrent garbage collector for servers
Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, Erez Petrank
November 2005 **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Volume 27 Issue 6
Publisher: ACM Press
Full text available: pdf(683.50 KB) Additional Information: full citation, abstract, references, index terms

Multithreaded applications with multigigabyte heaps running on modern servers provide new challenges for garbage collection (GC). The challenges for "server-oriented" GC include: ensuring short pause times on a multigigabyte heap while minimizing throughput penalty, good scaling on multiprocessor hardware, and keeping the number of expensive multicycle fence instructions required by weak ordering to a minimum.We designed and implemented a collector facing these demands building on th ...

Keywords: Garbage collection, JVM, concurrent garbage collection

8 Error-free garbage collection traces: how to cheat and not get caught
Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S. McKinley, Darko Stefanović
June 2002 **ACM SIGMETRICS Performance Evaluation Review , Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems SIGMETRICS '02**, Volume 30 Issue 1
Publisher: ACM Press
Full text available: pdf(105.06 KB) Additional Information: full citation, abstract, references, citings

Programmers are writing a large and rapidly growing number of programs in object-oriented languages such as Java that require garbage collection (GC). To explore the design and evaluation of GC algorithms quickly, researchers are using simulation based on traces of object allocation and lifetime behavior. The *brute force* method generates perfect traces using a whole-heap GC at every potential GC point in the program. Because this process is prohibitively expensive, researchers often use < ...

9 Tuning garbage collection for reducing memory system energy in an embedded java environment
G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, M. Wolczko
November 2002 **ACM Transactions on Embedded Computing Systems (TECS)**, Volume 1 Issue 1
Publisher: ACM Press
Full text available: pdf(740.23 KB) Additional Information: full citation, abstract, references, citings, index terms

Java has been widely adopted as one of the software platforms for the seamless integration of diverse computing devices. Over the last year, there has been great momentum in adopting Java technology in devices such as cellphones, PDAs, and pagers where optimizing energy consumption is critical. Since, traditionally, the Java virtual machine (JVM), the cornerstone of Java technology, is tuned for performance, taking into account energy consumption requires reevaluation, and possibly redesign of t ...

Keywords: Garbage collector, Java Virtual Machine (JVM), K Virtual Machine (KVM), low power computing

10 Connectivity-based garbage collection
Martin Hirzel, Amer Diwan, Matthew Hertz

October 2003 **ACM SIGPLAN Notices , Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications OOPSLA '03**, Volume 38 Issue 11
Publisher: ACM Press
Full text available: pdf(521.65 KB) Additional Information: full citation, abstract, references, citings, index terms

We introduce a new family of connectivity-based garbage collectors (Cbgc) that are based on potential object-connectivity properties. The key feature of these collectors is that the placement of objects into partitions is determined by performing one of several forms of connectivity analyses on the program. This enables partial garbage collections, as in generational collectors, but without the need for any write barrier.The contributions of this paper are 1) a novel family of garbage c ...

Keywords: connectivity based garbage collection

11 Garbage collecting the Internet: a survey of distributed garbage collection
Saleh E. Abdullahi, Graem A. Ringwood
September 1998 **ACM Computing Surveys (CSUR)**, Volume 30 Issue 3
Publisher: ACM Press
Full text available: pdf(337.65 KB) Additional Information: full citation, abstract, references, citings, index terms, review

Internet programming languages such as Java present new challenges to garbage-collection design. The spectrum of garbage-collection schema for linked structures distributed over a network are reviewed here. Distributed garbage collectors are classified first because they evolved from single-address-space collectors. This taxonomy is used as a framework to explore distribution issues: locality of action, communication overhead and indeterministic communication latency.

Keywords: automatic storage reclamation, distributed, distributed file systems, distributed memories, distributed object-oriented management, memory management, network communication, object-oriented databases, reference counting

12 Support for garbage collection at every instruction in a Java compiler
James M. Stichnoth, Guei-Yuan Lueh, Michał Cierniak
May 1999 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation PLDI '99**, Volume 34 Issue 5
Publisher: ACM Press
Full text available: pdf(1.06 MB) Additional Information: full citation, abstract, references, citings, index terms

A high-performance implementation of a Java Virtual Machine[1] requires a compiler to translate Java bytecodes into native instructions, as well as an advanced garbage collector (e.g., copying or generational). When the Java heap is exhausted and the garbage collector executes, the compiler must report to the garbage collector all live object references contained in physical registers and stack locations. Typical compilers only allow certain instructions (e.g., call instructions and bac ...

Keywords: Java, compilers, garbage collection

13 Using generational garbage collection to implement cache-conscious data placement
Trishul M. Chilimbi, James R. Larus
October 1998 **ACM SIGPLAN Notices , Proceedings of the 1st international symposium**

on Memory management ISMM '98, Volume 34 Issue 3
Publisher: ACM Press
Full text available: pdf(1.20 MB) Additional Information: full citation, abstract, references, citings, index terms

The cost of accessing main memory is increasing. Machine designers have tried to mitigate the consequences of the processor and memory technology trends underlying this increasing gap with a variety of techniques to reduce or tolerate memory latency. These techniques, unfortunately, are only occasionally successful for pointer-manipulating programs. Recent research has demonstrated the value of a complementary approach, in which pointer-based data structures are reorganized to improve cache loca ...

Keywords: cache-conscious data placement, garbage collection, object-oriented programs, profiling

14 Objects and their collection: The pauseless GC algorithm
Cliff Click, Gil Tene, Michael Wolf
June 2005 **Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments**
Publisher: ACM Press
Full text available: pdf(440.91 KB) Additional Information: full citation, abstract, references, index terms

Modern transactional response-time sensitive applications have run into practical limits on the size of garbage collected heaps. The heap can only grow until GC pauses exceed the response-time limits. Sustainable, scalable concurrent collection has become a feature worth paying for.Azul Systems has built a custom system (CPU, chip, board, and OS) specifically to run garbage collected virtual machines. The custom CPU includes a read barrier instruction. The read barrier enables a highly concurren ...

Keywords: Java, concurrent GC, custom hardware, garbage collection, memory management, read barriers

15 On the usefulness of type and liveness accuracy for garbage collection and leak detection
Martin Hirzel, Amer Diwan, Johannes Henkel
November 2002 **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Volume 24 Issue 6
Publisher: ACM Press
Full text available: pdf(684.85 KB) Additional Information: full citation, abstract, references, index terms

The effectiveness of garbage collectors and leak detectors in identifying dead objects depends on the *accuracy* of their reachability traversal. Accuracy has two orthogonal dimensions: (i) whether the reachability traversal can distinguish between pointers and nonpointers (*type accuracy*), and (ii) whether the reachability traversal can identify memory locations that will be dereferenced in the future (*liveness accuracy*). This article presents an experimental study of the impo ...

Keywords: Conservative garbage collection, leak detection, liveness accuracy, program analysis, type accuracy

16 Concurrency: Message analysis-guided allocation and low-pause incremental garbage collection in a concurrent language
Konstantinos Sagonas, Jesper Wilhelmsson
October 2004 **Proceedings of the 4th international symposium on Memory**

management
Publisher: ACM Press
Full text available: pdf(650.12 KB) Additional Information: full citation, abstract, references, index terms

We present a memory management scheme for a concurrent programming language where communication occurs using message-passing with copying semantics. The runtime system is built around process-local heaps, which frees the memory manager from redundant synchronization in a multithreaded implementation and allows the memory reclamation of process-local heaps to be a private business and to often take place without garbage collection. The allocator is guided by a static analysis which speculative ...

Keywords: Erlang, concurrent languages, incremental and real-time garbage collection, thread-local heaps

17 Concurrent garbage collection using hardware-assisted profiling
Timothy H. Heil, James E. Smith
October 2000 **ACM SIGPLAN Notices , Proceedings of the 2nd international symposium on Memory management ISMM '00**, Volume 36 Issue 1
Publisher: ACM Press
Full text available: pdf(1.74 MB) Additional Information: full citation, abstract, citings, index terms

In the presence of on-chip multithreading, a Virtual Machine (VM) implementation can readily take advantage of *service threads* for enhancing performance by performing tasks such as profile collection and analysis, dynamic optimization, and garbage collection concurrently with program execution. In this context, a hardware-assisted profiling mechanism is proposed. The *Relational Profiling Architecture* (RPA) is designed from the top down. RPA is based on a relational model similar ...

18 Creating and preserving locality of java applications at allocation and garbage collection times
Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, Jaswinder Pal Singh
November 2002 **ACM SIGPLAN Notices , Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications OOPSLA '02**, Volume 37 Issue 11
Publisher: ACM Press
Full text available: pdf(180.20 KB) Additional Information: full citation, abstract, references, citings, index terms

The growing gap between processor and memory speeds is motivating the need for optimization strategies that improve data locality. A major challenge is to devise techniques suitable for pointer-intensive applications. This paper presents two techniques aimed at improving the memory behavior of pointer-intensive applications with dynamic memory allocation, such as those written in Java. First, we present an allocation time object placement technique based on the recently introduced notion of *p* ...

***Keywords**: JVM, Java, garbage collection, heap traversal, locality, locality based graph traversal, memory allocation, memory management, object co-allocation, object placement, prolific types, run-time systems*

19 The measured cost of copying garbage collection mechanisms
Michael W. Hicks, Jonathan T. Moore, Scott M. Nettles
August 1997 **ACM SIGPLAN Notices , Proceedings of the second ACM SIGPLAN international conference on Functional programming ICFP '97**, Volume 32 Issue 8
Publisher: ACM Press

Full text available: pdf(1.65 MB) Additional Information: full citation, abstract, references, citings, index terms

We examine the costs and benefits of a variety of copying garbage collection (GC) mechanisms across multiple architectures and programming languages. Our study covers both low-level object representation and copying issues as well as the mechanisms needed to support more advanced techniques such as generational collection, large object spaces, and type segregated areas.Our experiments are made possible by a novel performance analysis tool, *Oscar*. Oscar allows us to capture snapshots of pr ...

20 Performance of a hardware-assisted real-time garbage collector
William J. Schmidt, Kelvin D. Nilsen
November 1994 **ACM SIGPLAN Notices , ACM SIGOPS Operating Systems Review , Proceedings of the sixth international conference on Architectural support for programming languages and operating systems ASPLOS-VI**, Volume 29 , 28 Issue 11 , 5
Publisher: ACM Press

Full text available: pdf(1.16 MB) Additional Information: full citation, abstract, references, citings, index terms

Hardware-assisted real-time garbage collection offers high throughput and small worst-case bounds on the times required to allocate dynamic objects and to access the memory contained within previously allocated objects. Whether the proposed technology is cost effective depends on various choices between configuration alternatives. This paper reports the performance of several different configurations of the hardware-assisted real-time garbage collection system subjected to several different ...

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next